Preprint
UCRL-JC-150544

Component Technology for Laser Plasma Simulation

W.J. Bosl, S.G. Smith, T. Dahlgren, T. Epperley, S. Kohn, G. Kumfert

This article was submitted to International Symposium on Computing in Object-Oriented Parallel Environments, Seattle, WA, November 3-5, 2002

June 17, 2002

Component Technology for Laser Plasma Simulation

William J. Bosl
bosl@llnl.gov

Steven G. Smith
sgsmith@llnl.gov

Tamara Dahlgren
dahlgren1@llnl.gov

Thomas Epperley
epperley@llnl.gov

Scott Kohn
kohn1@llnl.gov

Gary Kumfert
kumfert@llnl.gov

Lawrence Livermore National Lab
P.O. Box 808, L-561
Livermore, CA 94551

ABSTRACT
This paper will discuss the application of high performance component software technology developed for a complex physics simulation development effort. The primary tool used to build software components is called Babel and is used to create language-independent libraries for high performance computers. Components were constructed from legacy code and wrapped with a thin Python layer to enable run-time scripting. Low-level components in Fortran, C++, and Python were composed directly as Babel components and invoked interactively from a parallel Python script.

Categories and Subject Descriptors
D.2.12 [**Software Engineering**]: Interoperability – *distributed objects, interface definition languages.*

General Terms
Algorithms, Performance, Languages

Keywords
Components, scientific computing, numerical methods, physics.

1. INTRODUCTION
The scientific computing community has invested a significant amount of resources towards the development of high-performance scientific simulation software, including numerical libraries, visualization, steering, software frameworks, and physics packages. Unfortunately, because this software was not designed for interoperability and re-use, it is often difficult to share these sophisticated software packages among applications due to differences in implementation language, programming style, or calling interfaces. It is highly desirable to be able to reuse large and complicated software packages without having to devote large amounts of time to re-engineer them [1]. Moreover, many of the simulations that are required today involve multiple physical and chemical processes, so-called multiphysics simulations. Building these codes from pre-tested software components is much more reliable and efficient than trying to build a complete simulator from scratch [2].

One example of a complicated multiphysics simulation problem is the interaction of lasers with plasmas. Simulation of laser plasma interaction is an important design tool, complementing theoretical analysis and experimentation for developing complicated laser tools for studying inertial confinement fusion. The software required for simulating these complex physical processes reflects the physical system: it is complex. To carry out numerical experiments and analyze the resulting computational data, the software must be flexible enough to allow scientists to quickly and easily compare competing physics models and alternative design strategies. Constructing complex simulation codes from available software components is an efficient strategy for building a new laser plasma simulation code.

In this paper, will present our experiences wrapping a large scientific simulation code using the Babel language interoperability tool [8] so that the application could be driven from the Python scripting language. Furthermore, we were able to freely mix C++, Fortran, and Python modules in the software. For example, from the scripting layer, we were able to call the application code in C++, which in turn called a numerical routine written in Fortran, which in turn called a bounary condition routine written in Python. This language interoperability enabled us to rapidly prototype new boundary conditions modules in Python without recompiling or linking the whole code. We discovered that compiler incompatibilities introduced some difficulties in code reuse. This problem is ubiquitous and is not limited to the Babel tool. We will discuss the trade-offs using a tool such as Babel as compared to a more traditional wriapping solution such as SWIG.

2. ALPS: Adaptive Laser Plasma Simulator
The ability to predict and control laser-plasma interactions is critical for the design of inertial confinement fusion (ICF) experiments. ICF involves the use of high powered lasers to rapidly ionize and compress hydrogen fuel pellets sufficiently to initiate a fusion reaction. During these experiments, a plasma filled region is created by the ionizing fuel. The laser must continue to propagate through the plasma region to achieve the desired distribution of energy at the target fuel pellet. Simulation of the laser plasma interactions is used to predict and control laser parameters for ICF experiments.

The Adaptive Laser Plasma Simulator (ALPS) project [3] is being developed using the SAMRAI (Structured Adaptive Mesh Refinement Applications Infrastructure) [4,5] system currently under development in CASC. SAMRAI is a C++ class library that supports the development of application codes utilizing structured adaptive mesh refinement (AMR) algorithms. Parallelism on distributed memory architectures is handled by the framework, freeing the user from most of these details. Data layout and interprocess communication is performed through an interface to the standard Message Passing Interface (MPI) library.

3. Component Software Technology

Component technology is an extension of scripting and object-oriented software development techniques that specifically focuses on the needs of software re-use and interoperability. Component-based software techniques address issues of language independence and component connection behavior that other software techniques do not address. To use a hardware analogy, a component is like a "software integrated circuit" with well-defined pin-outs that may be connected to compatible pins on other "software integrated circuits." Figure 1 is a cartoon illustration of how we used Babel as the backplane to connect software components together to create an application.

3.1 Commercial solutions

Component approaches based on CORBA [9], COM [12], and Java technologies are widely used in industry but will not scale to support large parallel applications in science and engineering. Our research focuses on the unique requirements of scientific computing on high-performance machines, such as fast in-process connections among components, language interoperability for scientific languages, and data distribution support for massively parallel SPMD components.

3.2 Babel

Babel is a language interoperability tool that uses a *Scientific Interface Definition Language* (SIDL) to describe component interfaces. Using SIDL descriptions, Babel automatically generates code to mediate differences between components written in different languages.

Computational scientists developing large simulation codes often face difficulties due to language incompatibilities among various software libraries. Scientific software libraries are written in a variety of programming languages, including Fortran, C, C++, or a scripting language such as Python. Language differences often force software developers to generate mediating glue code by hand. In the worst case, computational scientists may need to re-write a particular library from scratch or not use it at all. We have developed a tool called Babel that addresses language interoperability and re-use for high-performance parallel scientific software. Its purpose is to enable the creation, description, and distribution of language independent software libraries.

Babel addresses the language interoperability problem using Interface Definition Language (IDL) techniques. An IDL describes the calling interface (but not the implementation) of a particular software library. IDL tools such as Babel use this interface description to generate *glue code* that allows a software library implemented in one supported language to be called from any other supported language. We have designed a Scientific Interface Definition Language (SIDL) that addresses the unique needs of parallel scientific computing. SIDL supports complex numbers and dynamic multi-dimensional arrays as well as parallel communication directives that are required for parallel distributed components. SIDL also provides other common features that are generally useful for software engineering, such as enumerated types, symbol versioning, name space management, and an object-oriented inheritance model similar to Java.

The Babel parser, which is available either at the command-line or through the Alexandria web interface, reads SIDL interface specifications and generates an intermediate XML representation. XML is a useful intermediate language since it is amenable to manipulation by tools such as a repository or a problem solving environment. XML interface descriptions are stored either in a local file repository or on the web using Alexandria. The vision is that a scientist downloading a particular software library from the component repository will receive not only that library but also the required language bindings generated automatically by the Babel tools.

The Babel code generator reads SIDL XML descriptions and automatically generates glue code for the specified software library. This glue code mediates differences among calling languages and supports efficient inter-language calls within the same memory address space and, eventually, across memory spaces for distributed objects. The code generators create four different types of files: stubs, skeletons, Babel internal representation, and implementation prototypes. The Babel internal object representation created by the code generators is similar to that used by COM, CORBA's Portable Object Adaptor, and scientific libraries such as PETSc. The internal object representation is essentially a table of function pointers, one for each method in an object's interface, along with other information such as internal object state data, parent classes and interfaces, and Babel data structures. Stub and skeleton code translates between the calling conventions of a particular language and the internal Babel representation. The code generators also create implementation files that contain function prototypes to be filled in by the library developers. To simplify the task of library writers, we have added automatic Makefile generation as well as a *code splicing* capability that preserves old edits during the regeneration of implementation files after modifications to the SIDL source. Finally, the run-time library provides general services such as reference counting and dynamic type identification. In the future, we expect to support dynamic loading of objects, reflection, and a dynamic invocation interface.

4. PyALPS

Currently, our laser plasma simulations are carried out using a uniform rectangular grid. This prohibits the use of high resolution in the regions of greatest interest by requiring a uniform grid over the entire domain. However, the code currently used for laser-plasma simulation is highly developed as a scientific and engineering design tool. In particular, an in-house scripting language called Yorick [11] is used for interactive steering and control of laser calculations. Yorick is an interpreted programming language, designed for postprocessing or steering large scientific simulation codes. Smaller scientific simulations or calculations can be written as standalone yorick programs. The language features a compact syntax for many common array operations, so it processes large arrays of numbers very efficiently.

4.1 Scripting

For use as a scientific and engineering design tool, ALPS requires the run-time flexibility of a scripting language, such as the Yorick capability that current laser physicists are accustomed to having. We adopted Python as a scripting language because it has a large and growing scientific user base and has a parallel implementation.

Since detailed simulations of laser plasma interactions can consume many hours of supercomputer time, it is often desirable to do calculations with either limited spatial resolution or a small number of time steps, then look at the results and determine whether some adjustment of the parameters is needed before continuing on with a lengthy calculation. Similarly, short period simulations may be used to examine the effects of parameter variations. Scripting enables laser scientists to perform simulations in a controlled fashion to maximize the amount of information that can be obtained in a limited time [8]. It also allows a great deal of flexibility by allowing different or new physics modules to be invoked quickly and easily. Scripted codes can be run interactively or in batch mode, giving the user considerable flexibility over a simulation.

We have used Babel to develop a scripted version of ALPS that uses Python as the scripting language. Wrapping parts of the ALPS code using Babel enables the creation of plug-n-play modules in a variety of supported languages. From the highest level at which users interact with pyAlps, the ALPS application appears to be a Python package, consisting of pure Python modules. that enables application users to compare ALPS results against those produced by an existing computational tool. The scripted interface will also allow ALPS users to interact with a running simulation to visualize data on-the-fly. This collaboration is the first to demonstrate Babel's applicability in a large-scale scientific application.

One of the primary goals of creating a scripted version of ALPS was to enable users to run ALPS interactively. Babel was used to create thin Python wrappers for important capabilities in the ALPS code. Specifically, we wrote interface files with Babel's Scientific Interface Definition Language (SIDL), which is similar to the IDL interface used to write CORBA interfaces. The SIDL file is a language-independent, object oriented description of the attributes (member variables) and methods associated with interfaces and classes. Babel uses the information in the SIDL file to create language bindings for any of the supported languages.

An example of a SIDL file is shown here. It contains class definitions for the basic Alps class and for beam modules, which compute the energy intensity contained in a laser beam. The SIDL file is used by the babel software to generate client-side and server-side code, each in a specified language. For the Alps class, the client is written in Python and all relevant files are presented to the user as the pyAlps package. Once imported as a Python package, an Alps class is created and methods can be invoked. After initialization from an input file or restart data file, the user may invoke several different run options in order to control time stepping precisely. Visualization files can be written at any point after the simulation has run to the currently-specified time and viewed using visualization software. Parameters can be adjusted using Python-wrapped database manipulation methods for the input variables.

The following code is an example of a SIDL file for the pyALPS package. Babel uses the information in this file to create glue code in any of the supported languages to wrap each of the specified objects.

```
version pyAlps 0.1;
package pyAlps {
  class Alps {
     void initialize(in pySAMRAI.InputDatabase database);
     void initializeFromRestart(in string dir, in int num, in
                          pySAMRAI.InputDatabase database);
     double run(in double time);
     double runToFinish();
     double runTo(in double time);
     double step(in int num_iter);
     double stepTo(in int iteration);
     void writeRestart(in string fname, in int seq_num_ext);
     void writeVis(in string fname, in int seq_num);
     void finalize();
  }
  abstract class Beam {
     abstract void setBeam0(inout array<dcomplex,2> amp);
     final void setDopplerShift(in double a_doppler_shift);
     final double getDopplerShift();
     final void setCenter(in array<double,1> a_center);
     final void getCenter(out array<double,1> a_center);
     final void setMaxIntensity(in double a_intensity);
     final void getMaxIntensity(out double a_intensity);
  }
  class Cos2_Beam extends Beam {
     void setBeam0(inout array<dcomplex,2> amp); }
  class SphericalCos2_Beam extends Beam {
     void setBeam0(inout array<dcomplex,2> amp); }
  class Gaussian_Beam extends Beam {
     void setBeam0(inout array<dcomplex,2> amp); }
  class SuperGaussian_Beam extends Beam {
     void setBeam0(inout array<dcomplex,2> amp); }
  }
```

In particular, note that the beam class is declared to be an abstract class. This means that at least of the member functions of the beam class is abstract and is not defined within the beam class. Subclasses of the general beam class must define a setBeam0 method. The abstract beam class also declares a number of member functions that will be explicitly defined in the implementation of the beam class. These member functions are common to all subclasses of the beam module, although they may be substituted with new functions in subclasses. Babel can create Beam modules in any of the supported languages, currently including F77, Python, C, and C++ from the SIDL file.

Our initial task was to decompose the ALPS code into components that were appropriate for run-time scripting. The primary tasks performed in the monolithic code were to read and process input data, initialize data structures, loop through a specified time loop, and output data at regular intervals in the time loop. These code segments formed the basic components that were to be controlled from the script.

The ALPS code simulates the interaction of a set of laser beams with a plasma in space and time. The computational grid is a sophisticated adaptive, multilevel grid that is required for high resolution. Often, run-time parameters for the complex simulation runs are not know precisely. Scientists needed a simulation tool that could be run a certain number of time steps, stopped and queried using visualization tools to inspect intermediate field variables, then modified by changing certain key parameters and run forward in time for a fixed interval again. This gave the scientists a steering capability through Python scripting.

A parallel version of Python, pyMPI, developed at LLNL and available publically through SourceForge [12] was adopted for Python scripting. ALPS is built using the SAMRAI framework for adaptive mesh simulations on parallel machines, together with legacy Fortran code obtained from laser physicists. Linear solvers from the PETSc library and HYPRE are available through the SAMRAI framework and invoked for solving linear systems. Babel was able to generate code to glue together all these packages in appropriate components. Of particular note was the decomposition of SAMRAI into components for data I/O and mesh initialization. From the scientist's view, pyALPS looks like a normal python script. An example of a pyALPS script is shown here:

```
import sys
import pySAMRAI.InputDatabase
import pySAMRAI.Alps

# Create the input database
inputdb = pySAMRAI.InputDatabase.InputDatabase()
inputdb.initialize("ALPS")
inputdb.parseInputFile("alps.input")

# Create alps object and initialize the state
alps = pySAMRAI.Alps.Alps()
alps.initialize(inputdb)

# Change some values
griddingdb = inputdb.getDatabase("GriddingAlgorithm")
print("Old efficiency_tolerance = %f" %
griddingdb.getDouble("efficiency_tolerance"))
print("Old combine_efficiency = %f" %
griddingdb.getDouble("combine_efficiency"))
griddingdb.putDouble("efficiency_tolerance", 0.90)
griddingdb.putDouble("combine_efficiency", 0.90)
print("New efficiency_tolerance = %f" %
griddingdb.getDouble("efficiency_tolerance"))
print("New combine_efficiency = %f" %
griddingdb.getDouble("combine_efficiency"))

# Step 5 time steps ...
alps.Step(5)
# ... then do something with the data!
# Run to the end specified in the input file
alps.runToFinish()
# Finalize everything
alps.finalize()
```

One of the primary difficulties encountered in this project was related to the need to create dynamic libraries for run-time loading. Incompatible compiler options seemed to cause the most build problems. During the integration process the low level details of simply building the code caused an unexpected number problems. Several of the packages we were integrating had not been compiled as a shared library before. This mandated a reworking of the build systems in order to support the necessary compilation steps. While this was expected, the brittleness of the build process was not. We found that even slight variations in the compiler options used to compile each package could cause link or runtime failures.

The runtime failures in particular are troublesome since a method invocation would fail in a system library for no obvious reason. To overcome this we standardized on a set of compilers and compile flags for all packages. While this is a simple (and obvious) solution, it is not a satisfactory solution if the goal is to have a large set of easy to use components for widespread use. Given the target audience for a scientific component architecture contains developers for whom dynamic linking will be a new experience, these types of problems could pose a barrier for software reuse, especially for software in object or component form. The component software community may need to move towards some kind of compiler meta-data for packages or something else to facilitate mixing of binary libraries, especially with C++.

Creating SIDL files needed to wrap each of the components is is a little tedious, but is relatively straight-forward. We did not find this to be a particularly difficult issue.

4.2 Plug and Play Modularity

In addition interactive control of simulations, the capability of easily swapping in alternative physics modules is a desirable new feature for laser plasma simulations. Scientific investigation using simulation often involves testing and comparing alternative physics modules or new algorithms. Our goal was to enable rapid replacement of classes, subroutines, or groups of related classes and subroutines with alternatives.

To do this, appropriate pieces of code were wrapped using Babel and made into Babel components. These components can be accessed by driver routines written in any of the Babel supported languages. Alternative components can then be written by application scientists in any language that's convenient and wrapped with Babel to make an alternative component that can be seamlessly interchanged with the original component. Because the application scientist is free to implement new components in a language such as Python, new algorithms can be written quickly and tested in the pyALPS code. Important components can be optimized in another programming language later if desired.

One of the novel and powerful capabilities provided by Babel components is the ability to call any of the supported languages from any other. Thus, not only can Python call C or Fortran subroutines as, for example, SWIG extensions to Python, but Fortran can also call Python functions. We used this feature to create a powerful plug and play capability for scientific exploration of new beam modules.

4.2.1 Beam Modules

In the ALPS code, beam calculations are invoked from within the legacy Alps code. The original beam subroutines are written in

Fortran and are called from Fortran subroutines, which are originally invoked from the Alps C++ driver code. Using the SIDL file shown above for the Beam class, we made Beams a component of the system and modified the ALPS driver to call Beam components rather than the original embedded Fortran subroutines. Beam modules clients were created in Fortran using Babel to enable us to use the original Fortran beam calculations. Once this was done, we also created Python beam clients to demonstrate this capability. The advantage of Python beam modules is that they can be created quickly and do not need to be compiled to be invoked by the pyAlps simulator. This provides a versatile tool for scientific experimentation.

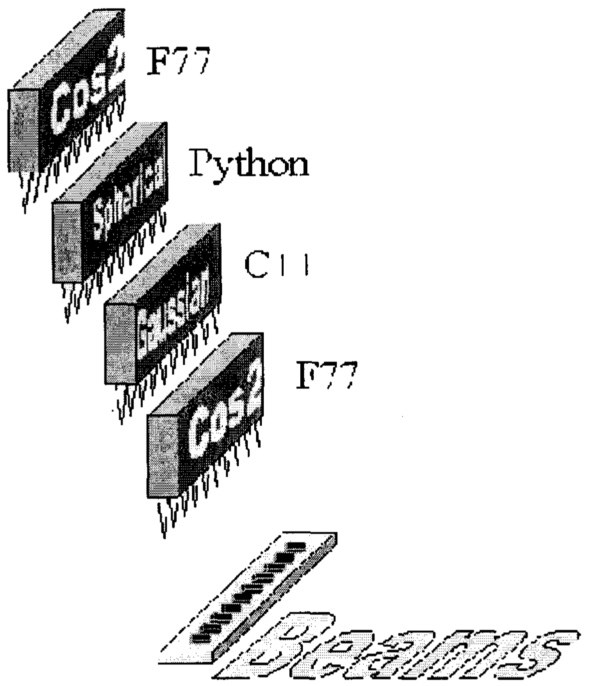

Figure 1. Physics components can be written in any of the languages supported by Babel. Components written in Python, for example, can be invoked without recompiling to rapidly test new algorithms.

4.2.2 Lessons Learned

Creating new modules was not as difficult as building the components created from legacy code and linking them together. This is due largely to the fact that new beam modules are designed and written specifically for the component system. Python modules are particularly easy to write and invoke from the Python script. Perhaps the only difficulty in adopting this approach was learning to use Babel arrays within Fortran in order to pass them to the component layer on the client side. Arrays must be passed back and forth to client and server in a language independent fashion and this is accomplished by requiring the creation of Babel arrays in all user code.

5. Discussion

Several different approaches are available today to build language independent components that can be re-used in multiple applications, used to assemble complex multi-physics simulators from pre-built software, and run simulation codes from a scripting language such as Python. Babel is a tool that offers certain unique features if those features are required, including a powerful array syntax, support for complex numbers, and parallel computing. The price for this capability is a need for careful attention to compiler options for all codes that must interoperate and the need to learn Babel data structures and the Babel scientific interface definition language. If Babel's unique features are required, then this is a price that has to be paid, for there are few other options at this time that provide all these features.

6. ACKNOWLEDGMENTS

Funding for this project is provided by the LLNL Laboratory Directed Research and Development program and the DOE Office of Science.

7. REFERENCES

[1] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois Mcinnes, Steve Parker, and Brent Smolinski, "Toward a Common Component Architecture for High Performance Scientific Computing," *High Performance Distributed Computing Conference*, 1999.

[2] A. Cleary, S. Kohn, S. Smith, B. Smolinski, "Language Interoperability Mechanisms for High-Performance Scientific Applications," *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, Yorktown Heights, NY, October 21-23, 1998.

[3] M. Dorr and X. Garaizar, *The ALPS Home Page*, http://www.llnl.gov/CASC/alps.

[4] R. Hornung and S. Kohn, "The Use of Object-Oriented Design Patterns in the SAMRAI Structured AMR Framework," *Proceedings of the SIAM Workshop on Object-Oriented Methods for Inter-Operable Scientific and Engineering Computing*, October 1998. See http://www.llnl.gov/CASC/SAMRAI.

[5] R. Hornung and S. Kohn, "Managing Application Complexity in the SAMRAI Object-Oriented Framework," *Concurrency and Computation: Practice and Experience* (special issue on Software Architecture for Scientific Applications), 2001.

[6] S. Kohn, G. Kumfert, J. Painter, and C. Ribbens. "Divorcing Language Dependencies from a Scientific Software Library," *Proceedings of the SIAM Conference on Parallel Processing for Scientific Computing*, 2001.

[7] J. Ousterhout, *Scripting: Higher Level Programming for the 21st Century*, **IEEE Computer**, March 1998.

[8] *CORBA Components*, Object Management Group, OMG TC Document orbos/99-02-95, March 1999. See http://www.omg.org

[9] B. Smolinski, S. Kohn, N. Elliott, and N. Dykman, "Language Interoperability for High-Performance Parallel Scientific Components," *International Symposium on Object-Oriented Parallel Environments (ISOPE)*, December 1999.

[10] See http://sourceforge.net/projects/pympi.

[11] *The Yorick Home Page*, 2001. See ftp://ftp-icf.llnl.gov/pub/Yorick/yorick-ad.html.

[12] http://www.microsoft.com/com/default.asp.